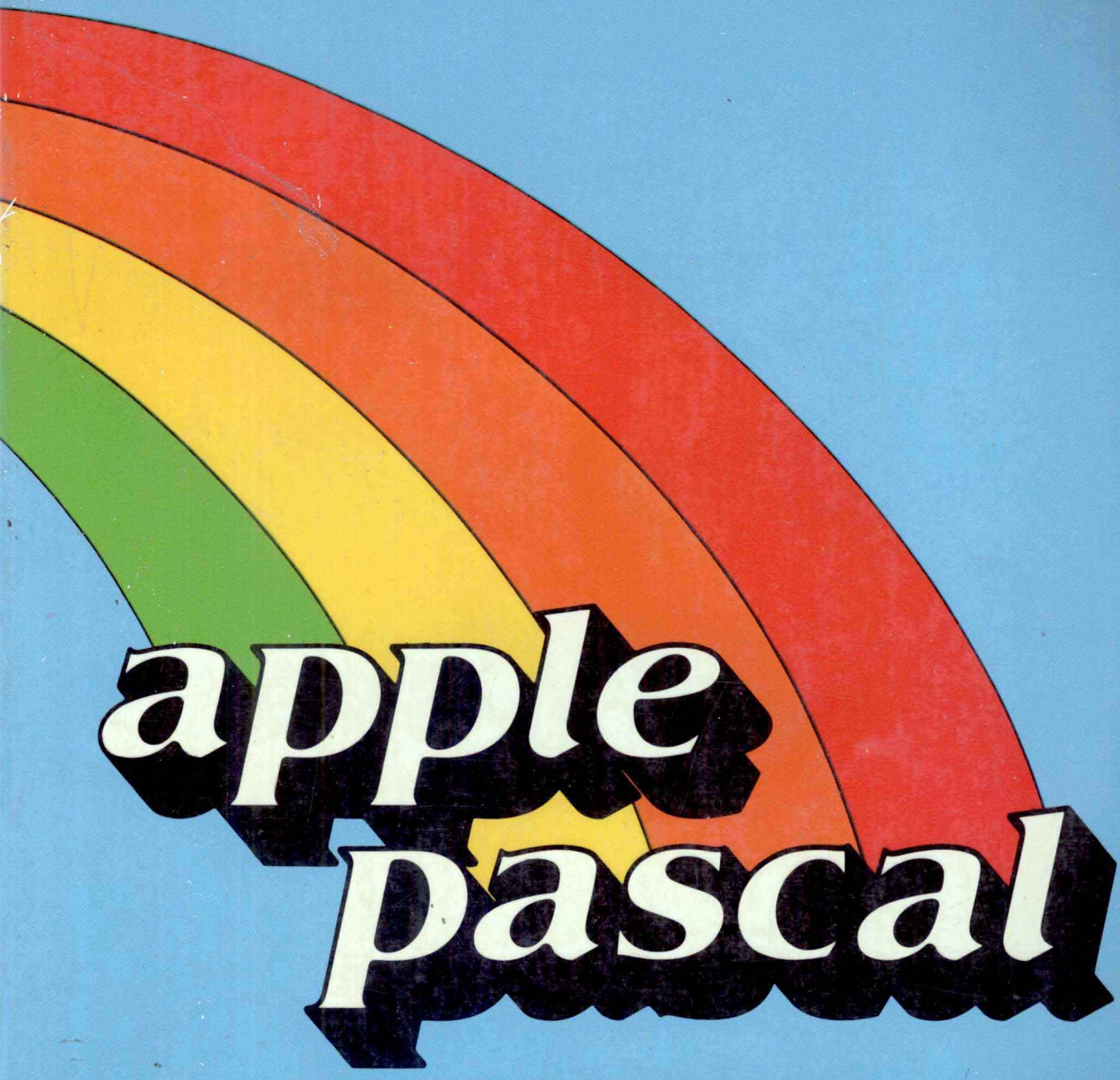


apple pascal

A PROGRAMMING GUIDE

Allen B. Tucker, Jr.

Apple Pascal
A Programming Guide

ALLEN B. TUCKER, JR.

Computer Science Program
Georgetown University

HOLT, RINEHART AND WINSTON

New York Chicago San Francisco Philadelphia
Montreal Toronto London Sydney Tokyo
Mexico City Rio de Janeiro Madrid

The Apple Pascal system incorporates UCSD Pascal™ and Apple extensions for graphics, sound, paddles, and other functions.

"Apple" and "Apple Pascal" are trademarks. "Apple" is a registered trademark of Apple Computer, Inc. "UCSD Pascal" is a trademark of the Regents of the University of California. Unauthorized use of these trademarks is contrary to the laws of the State of California and of the Federal Government.

This book is a tutorial guide to Apple Pascal, not a formal specification of the software as delivered to the buyer now or in the future software revisions. Apple Computer, Inc. makes no warranties with respect to this book or to its accuracy in describing any version of the Apple Pascal software product.

Address correspondence to:
383 Madison Ave. New York, NY 10017

Library of Congress Cataloging in Publication Data

Tucker, Allen B.
Apple Pascal

Includes index.
1. Apple computer—Programming. 2. Apple II (Computer)—Programming. 3. PASCAL (Computer program language) I. Title.
QA76.8.A66T83 001.64'2 82-912
ISBN 0-03-059547-9 AACR2

Printed in the United States of America
2 3 039 9 8 7 6 5 4 3 2

CBS COLLEGE PUBLISHING
Holt, Rinehart and Winston
The Dryden Press
Saunders College Publishing

Preface

Pascal has recently emerged as an important language for teaching computer science concepts and programming methodologies. It is available on a wide variety of computers, including some microprocessors. Pascal is a language rich in expressive power, applicable to a wide variety of uses, and yet unusually simple and consistent in its syntax and semantics.

The purpose of this book is to provide a unified introduction to programming and Pascal, from the viewpoint of the Apple Pascal system. This system has become very popular in recent years. We hope that this book will reach the wide variety of interests represented by those who use Apple computers. This book encourages self-paced learning and frequent reinforcement of concepts by hands-on programming lab exercises.

These lab exercises are grouped into three different subject areas, which we call business, math/science, and general. They are organized so that the readers may consolidate their understanding of any particular programming concept by choosing a lab exercise in a subject area appropriate to their own interests. For instance, LAB05 tests the reader's mastery of elementary loops, using the WHILE construct. The reader may choose among LAB05B (the business problem), LAB05M (the math/science problem), or LAB05G (the general problem) to demonstrate that mastery. LAB05B asks for a program that prints an interest payment table; LAB05M asks for a program that prints a list of factorials; and LAB05G asks for a program that averages *m* numbers. All three require an elementary loop.

This book also reflects our firm belief that a language is best taught by first introducing a subset that will allow the reader to solve several elementary problems and master basic programming concepts. We have defined a subset of Pascal and dubbed it ESP, Eight Statement Pascal. The eight statements of ESP are fully described and illustrated in Chapter 2, and summarized in Appendix G at the end of the book. We think that the acronym ESP is particularly appropriate for an introduction to programming, since there is a kind of sixth sense associated with this delicate art.

We also believe that any complete introduction to programming should provide a selection of programming problems and topics that truly reflect the variety of ways in which computers are being used today. To satisfy this requirement, we have collected 60 different programming lab problems (20 in each of the aforementioned subject areas). Of these, LAB01 through LAB06 test essential programming skills and the features of the

different programming lab problems (20 in each of the aforementioned subject areas). Of these, LAB01 through LAB06 test essential programming skills and features of the SSB subset language. LAB07 through LAB20 are individually associated with the following topics:

- Arrays
- Subprograms
- Formatted input/output and graphics
- Cross-tabulation and statistics
- Simulation of board games
- File processing
- Natural-language text processing

Thus, the lab problems provide a most varied and representative selection of computer applications for the uninitiated reader. This broad selection also serves to demonstrate the versatility of BASIC as a programming language.

The individual chapters should be covered in order. LAB01 through LAB06 should be done while covering Chapter 2, while LAB07 through LAB20 are keyed to individual Chapters 4 through 9 in the following manner:

Chapter		*Lab*
4	Simple Arrays	07–09
5	Functions and Procedures	10–12
6	Input/Output Options and Graphics	13, 14
7	Multidimensional Arrays	15, 16
8	Files and Records	17, 18
9	Character Strings	19, 20

There are also exercises, for drill and practice with different elements of BASIC syntax and program tracing, at appropriate points in the book. Answers to many of these exercises are provided in Appendix F. No answers are provided for the lab problems themselves, although helpful hints are given with some.

Most texts which introduce BASIC tend to avoid any hardware-specific details, leaving the reader to learn for himself or herself the extensions and idiosyncrasies of a particular BASIC system. We take a different point of view on this matter. To properly introduce program development, which is the main subject of Chapters 1 and 3, we cannot avoid system-dependent features, such as "how to change a line of program text." Thus, we have integrated all aspects of creating and modifying a BASIC program *on the Apple computer* into our discussions of program development. Similarly, we have included topics such as "how to implement random file access" in the discussion of file processing in Chapter 8, so that the reader will appreciate this very important pro-

APPLE2:) for running Pascal programs. These steps, as well as the diskette initialization procedures, are simple and are described in the Pascal reference manual that accompanies the Apple computer.

Many persons have contributed to this book in different ways. I would especially like to thank Georgetown students Jeff Grant and Jay Bellwoar for their creative and critical evaluations of this manuscript in its various stages. Jeff also typed and proofread the manuscript, while Jay also designed many of the programming lab problems. I am grateful also to the reviewers, particularly Ray Geremia, whose advice has helped improve the programs and the pedagogy throughout. Finally, and most importantly, I want to thank my family — Maida, Jenny, and Brian — for their continual love and support as always.

Allen B. Tucker, Jr.

Contents

Chapter 1

The Computing Process

Computers have a "static" aspect and a "dynamic" aspect. The static aspect consists of the components, while the dynamic aspect represents the actual movement and manipulation of data that occurs when the components are activated. Both the static and the dynamic aspects of computers must be clearly understood in order to master the art of programming itself.

1.1 Apple Computer Organization

We first describe the static aspect of computers, which is known as *computer organization*. Five general components comprise the organization of a computer, as pictured below:

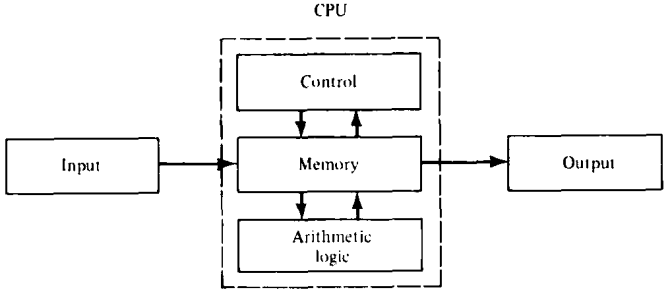


Figure 1–1. The Components of a Computer

In the center of Figure 1–1 is the computer's *central processing unit,* or CPU. Leading into the CPU from the left is the *input,* and leading out to the right is the *output.* The CPU itself has three parts; the *memory,* the *control,* and the *arithmetic/logic* circuitry.

The arrows that connect these components denote paths through which information can flow. That is, information can flow from the input to the memory, from the memory to the output, and in either direction between memory, control, and arithmetic/logic.

Figure 1–2 shows a picture of an Apple computer, with its five basic components identified. Here, the reader can get an idea of what these components actually look like.

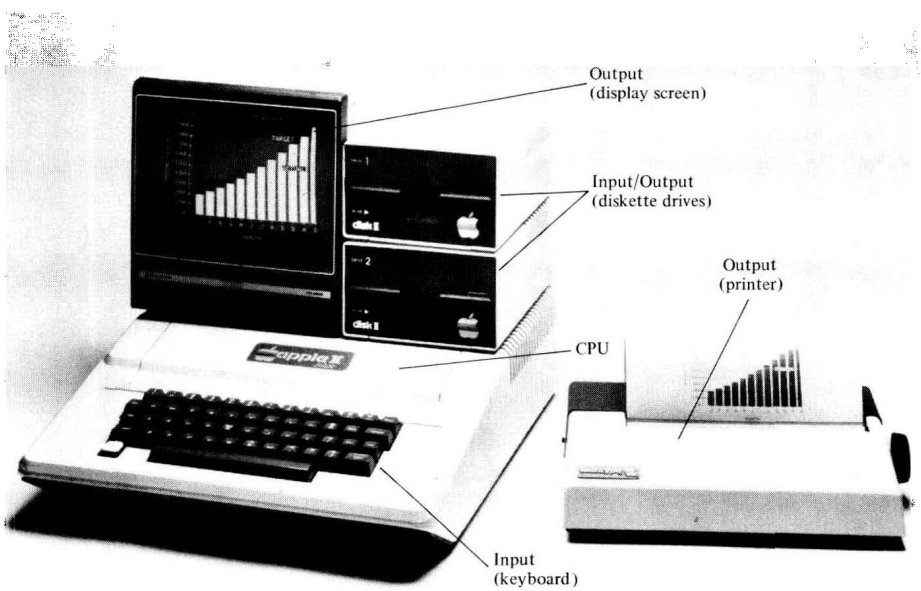


Figure 1–2. Components of an Apple Computer. (Courtesy of Apple Computer Inc.)

The *memory* of a computer holds two kinds of information, the *program* and some *data*. The *program* itself consists of à series of instructions that tells exactly what steps to perform. These instructions are actually carried out, or "executed," by the *control unit*. Some of the instructions tell the control to transfer information from the input to the memory; this is known as a "read" operation. Others tell the control to transfer information from the memory to the output; this is known as a "write" operation. Still others tell the control to perform an arithmetic operation (e.g., addition) or a comparison of two data values (e.g., to see which is greater). These kinds of operations are actually carried out by the *arithmetic/logic* part of the CPU.

Now, the actual *input* and *output* information can be represented in any of several different "computer-readable" media, including punched cards, an interactive terminal, video display, printed paper, magnetic tape, magnetic disk, and so forth. Shown with the Apple in Figure 1–2 are two magnetic disk units, an interactive terminal, video display, and a line printer. Each magnetic disk unit holds one cartridge, called a "diskette," which

may contain programs and data. Diskettes may be interchangeably mounted on the disk drive, but at any one time only one diskette may be present.

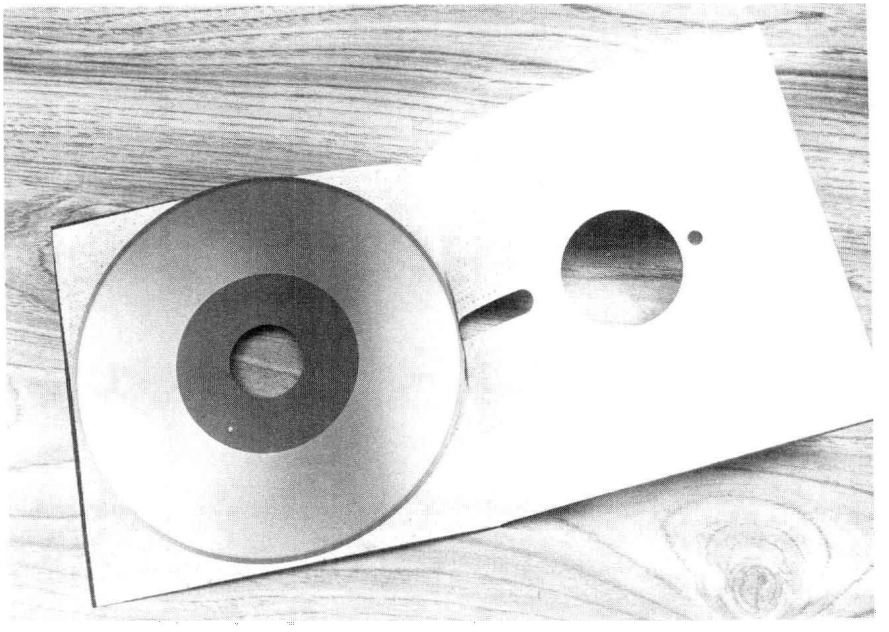


Figure 1–3. A Diskette.

1.2 Programs

The *program* is a sequence of instructions which, when executed by the control unit, defines exactly what should be done with the input data in order to produce a particular output. That is, the program specifies the "dynamic" aspect of the computer. Without the program, the components described in the foregoing section would be just a passive collection of hardware.

Functionally, the program is like a recipe; followed precisely, the recipe will yield the desired result. Yet, a program must be precisely and, sometimes, excruciatingly specified in order to fully define the task to be performed. Also, like a recipe, the program must be written in a very exact syntactic form, in order to be understood and properly executed. This form is known as the *programming language*.

There are many different programming languages in use today, such as ALGOL, COBOL, FORTRAN, PL/I, APL, SNOBOL, LISP, and Pascal. Each has its special strengths in one of the wide variety of application areas where programmers are working. In this book, we shall teach the programming language Pascal, because it is widely known, encourages good programming style, is easily taught and learned, and can be effectively used in the various programming situations that occur in mathematics, science, business, the humanities, government, and personal computing.

Because Pascal is a rather extensive language, we shall first teach an elementary part of it, so that the reader may master basic programming techniques before proceeding to advanced material. We have dubbed that elementary part as "Eight Statement Pascal," or ESP for short, which is the subject of Chapter 2. Additional Pascal features will be described, illustrated, and exercised in later chapters.

1.3 Program Development: An Overview

Although we teach the programming language Pascal, we have a far more important purpose in this book. That is, we shall introduce and teach the elements of *program development*. It is one thing to follow a recipe successfully and end up with an edible cake. But it is quite another to design and correctly describe the recipe in the first place.

More precisely, *program development* has as its purpose to design and demonstrate the correct functioning of a (Pascal) program that carries out a prescribed task. Examples of typical "prescribed tasks" are the following:

A. Add two numbers and display the resulting sum, given the original two numbers.
B. Compute the average of all three tests taken by each student in a class of 25, given the original 75 individual test scores (3 per student).
C. Translate a text from Spanish into English, given the original Spanish text.

As the reader can see, these examples range in difficulty from trivial to complex. Thus is the domain of program development. In this book, most of the program development tasks are like that of Example B; not trivial but achievable in a reasonable amount of time.

We prescribe these tasks as so-called "labs," numbered LAB00 through LAB20. LAB00 will be presented, programmed, and discussed in its entirety in this chapter; in fact, LAB00 *is* Example A given above. The labs are organized into three general subject area groups: "business," "math/science", and "general". Readers are encouraged to select labs that correspond with their subject-area interests. Labs are coordinated so that, for instance, LAB13B (that is, a business task) and LAB13M (that is, a math/science task) exercise the *same* Pascal features and program development techniques. (The suffix B, M, or G affixed to the LAB number identifies its subject-area as business, math/science, or general, respectively.)

Returning to the question of *program development*, this process can be subdivided into the following sequence of distinct steps:

1. Problem specification
2. Algorithm design
3. Program coding
4. Program preparation
5. Program execution
6. Program diagnosis and error correction

The following paragraphs describe each of these steps, using the above Example A for illustration.

1. *Problem Specification:* A clear and concise statement of the programming problem to be solved is, of course, a prerequisite to the development of the program itself. Recall the problem statement of Example A:

> add two numbers together and display the resulting sum, given the original two numbers.

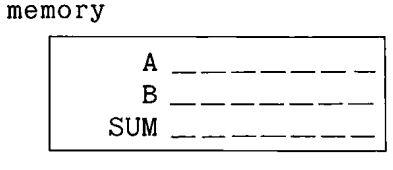
There is a kind of innate tedium in any such problem statement, which is due to the requirement for precision and completeness. The statement must always be reflective of the general capabilities and limitations of computers and programming. Moreover, the problem statement must be totally clear and fully comprehensible to the person who will write the program.

In this book, the programming problem statements are already developed, in the form of LAB00 through LAB20. Our purpose here is to teach programming and problem solving skills, rather than to teach the development of problem statements themselves. The area of computing in which problem statements are developed is known as *systems analysis and design.*

One element of a good problem statement is that it not only portrays the programming task to be performed (e.g., "add two numbers" in Example A) but also identifies the input data (e.g., "given the original two numbers") and the desired output (e.g., "dispay the resulting sum").

2. *Algorithm Design:* Here, the programmer translates the problem statement into a precise description of how the computer program will solve the problem. In general, algorithm* design begins with a sketch, in English, of the sequence of steps that the computer should follow to solve the problem. At this point, the programmer identifies all memory locations, known as *variables,* that are necessary for the program to perform properly. A memory location can be visualized as a place within the computer's memory which can hold a single data value, such as a number or an alphabetic character. A variable can be visualized as a memory location which is associated (by the program) with a unique name, such as A or SUM. For instance an algorithm design for Example A can be given as follows:

i. Identify A and B as the variables which will contain the two numbers to be added, and SUM as the variable which will contain their sum, as shown in the following picture of memory:



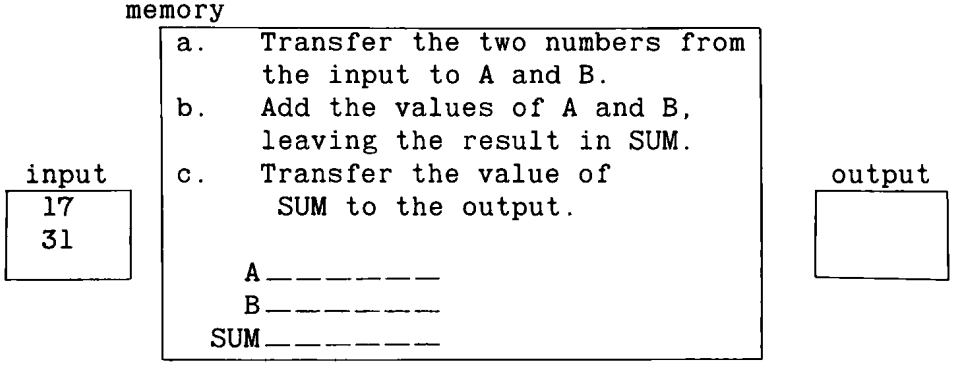
*The term "algorithm" means "a precise description of a computing task which will terminate in a finite number of steps." That description can be done in any suitable language, such as English or any programming language (e.g., Pascal, FORTRAN, COBOL, PL/I, BASIC, . . .). When done in a programming language, the algorithm is known as a "program."

ii. The sequence of steps required to solve this problem are:
   a. transfer the two numbers from the input to variables A and B respectively.
   b. Add the values of A and B, leaving the result in SUM.
   c. Transfer the value of SUM to the output.

An algorithm design always presumes certain overall operational or mechanical characteristics of computer programs:

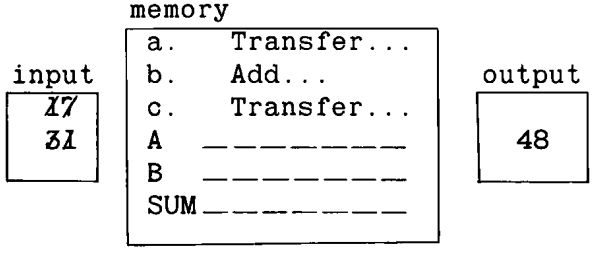
- that the input values must be brought into specific memory locations, or variables, before any arithmetic or other operations can be performed with them.
- that the computer's control unit carries out the individual steps of a program in the order in which they are written.
- that the result(s) of a program must be transferred from memory to the output in order for it to be displayed. The memory is an electronic medium, hidden from view. The input and output (e.g., a terminal screen or a printer) provide visible representations for data values and results.

To illustrate, the following diagram shows a complete configuration of input data, program, variables, and output immediately *before* steps a, b, and c of the algorithm are carried out by the computer's control unit:



Here, the algorithm is given 17 and 31 as two sample input values. In general, an algorithm is designed to handle *any* input that is suitable to the problem statement (e.g., any pair of numbers, in the case of Example A).

After the control has carried out all three steps specified by the algorithm, the resulting configuration will be as shown below:



In executing step a, the input numbers 17 and 31 are transferred to variables A and B, as shown. This is known as a "read" operation. In executing step b, the values of A and B are added together and the result is then stored in SUM, as shown. This is known as an "arithmetic operation," followed by an "assignment" of the result to SUM. In executing step c, the value of SUM is copied to the output (terminal or printed paper) as shown. This is known as a "write" operation.

Note that, in a read operation, the input values are transferred rather than copied. This implies that they are no longer available to be read in a later step. In a write operation, however, the variable's value is copied to the output medium, thus leaving the value intact in memory for use by a subsequent step in the program. Finally, all arithmetic operations and assignments *must* use values that are already in memory (e.g., by virtue of a previous read operation). Thus, the input value 17 shown above cannot directly take part in an arithmetic operation, but the value of the variable A (which is 17) can!

3. *Program Coding:* After the algorithm is fully designed, the program can be written, or "coded," in a programming language (such as Pascal). Although we have given the impression in previous sections that computers can execute English-language programs directly, the truth is that they cannot. Instead, they can understand only statements that are written in a programming language, and no others. The following is a complete Pascal program for the algorithm that was developed in previous sections:

```
PROGRAM P;
   VAR A, B, SUM : INTEGER;
   (* THIS PROGRAM SUMS TWO INTEGERS *)
   BEGIN
      READLN (A,B);
      SUM:=A+B;
      WRITELN (SUM)
   END.
```

The first line identifies the program, while the second line defines the variables needed by it. The third line is a "comment," and is inserted for documentation purposes. It has no effect on the actual execution of the program. Comments, in general, can be inserted anywhere within a Pascal program, provided they are enclosed in (* and *). The three lines between BEGIN and END describe the three steps required to accomplish the task—namely, a read operation, an arithmetic and assignment operation, and a write operation. These will be more fully explained in the paragraphs below.

4. *Program Preparation:* After the program has been coded, it is then entered at the terminal. Program preparation is fully described for the Apple in Section 1.5 below.

5. *Program Compile and Execution:* Before the computer actually executes the steps of the Pascal program, a "compiler" first scans the program's text to be sure that the steps are properly written (i.e., syntactically correct) and that the entire text represents a complete Pascal program. If not, it will note all syntactic errors and not proceed into actual execution of the program. If the program is correct, it is translated by the compiler from Pascal into the language of the computer on which it is to be executed. In the Apple vernacular, the original Pascal program is known as a "TEXT file," while its machine

language version is known as a "CODE file." This whole activity is known as the "compile step" for a program, and is fully explained in Section 1.6 below.

If the program is correct as determined by the compile step, with no syntactic errors, the computer will proceed to execute the resulting CODE file and produce output. This activity is known as the "execution" step for a program. Below are the results of executing the program shown above:

```
17      31
48
```

The first two numbers, 17 and 31, are the input numbers that are typed at the terminal in response to the READLN statement. The third number, 48, is the output produced by the WRITELN statement at the time it was executed. Note here that these three numbers which appear on the screen are not very informative by themselves. Later, we shall discuss how to refine programs, using "prompts" and other documentation, so that what appears on the screen is truly informative. At the moment, we are concentrating only on the basic programming, compiling, and execution process by itself, and these additional details would tend to confuse matters at this early stage in the book.

6. *Program Diagnosis and Error Correction:* The correct results may not necessarily be achieved by the first run of a program. More typically, two, three, or more runs are needed before the desired output is achieved. Three different types of errors can be made in the process:

   i. Syntactic, or "compile-time," errors—these result from failing to write Pascal statements properly, and are noted during the compile step of the run.
   ii. "Execution-time" errors—these occur during the execution step, and are noted by the computer in the output of the run.
   iii. Design errors—These errors result from writing a syntactically correct program, having it run, but achieving output that is incorrect, according to the original problem statement. These are not noted at compile time or execution time, since the computer has no way of knowing the original problem statement.

To illustrate, suppose we had incorrectly written the Pascal statement "SUM:=A+B" as "SUM:=A+" instead. This compile-time error would have been noted as shown below:

```
ERROR IN <FACTOR> (BAD EXPRESSION)
```

Thus, a compile-time error is easy to detect and diagnose, since the message is mnemonic and the cursor is placed at the position in the program display where the error occurred.